IBM TJ Watson Research Center - Advanced Compiler Technologies

Towards a portable OpenMP data sharing implementation for NVIDIA accelerators in the CLANG/LLVM toolchain

Gheorghe-Teodor Bercea

IBM Research

Team members:
Carlo Bertolli, Hyojin Sung,
Arpith C. Jacob, Alexandre Eichenberger,
Tong Chen, Kevin O'Brien.

What this talk is about

IBM

- Introducing a **new,** “upstream-able” data sharing scheme for CLANG/LLVM trunk (not to be confused with the current implementation).
- In the current talk we cover only the **first level of sharing**: from one thread in an OpenMP team to the rest of the threads in the team.
- Overcoming the problem that:

“In certain use cases, OpenMP’s **default sharing of local variables** is incompatible with the **default allocation into local memory** of local variables on NVIDIA GPUs.”

Mapping OpenMP to GPUs

IBM

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;
    }
  }
}
```

OpenMP allows nesting of regions with different numbers of threads.

Mapping OpenMP to GPUs

IBM

OpenMP semantics

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;                    1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;         all threads
    }
  }
}
```

Mapping OpenMP to GPUs

We need to share "c"

```
void test(){
  int c = 5000;
  #pragma omp target
  {
   c += 1;                    1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;           all threads
    }
  }
}
```

Mapping OpenMP to GPUs

IBM

Default NVPTX backend policy: "c" is allocated onto the thread local stack

```
void test(){
  int c = 5000;
  #pragma omp target
  {
   c += 1;                    1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;           all threads
    }
  }
}
```

Mapping OpenMP to GPUs

Default NVPTX backend policy: "c" is allocated onto the thread local stack

```
void test(){
  int c = 5000;
  #pragma omp target
  {
   c += 1;                      1 thread
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;             all threads
    }
  }
}
```

On GPUs threads cannot share a variable allocated on the local stack.

Function outlining

- In general: **OpenMP regions** delimited by different constructs will be outlined.
- The master thread assigns those regions to workers **dynamically: we therefore avoid dynamic thread launch in favour of dynamic work allocation to existing threads**.
- Outlining ensures that all parallel OpenMP regions have access to all the worker threads including OpenMP regions that are defined in other compilation units.
- **Data must be shared across multiple functions.**

OpenMP outlined regions example

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;                    MASTER
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;            WORKERS
    }
    c += 2;                    MASTER
  }
}
```

Changes to CLANG and the runtime

- The runtime maintains a list of references to the shared variables.
- The MASTER region needs to initialize this list.
- The WORKER region retrieves the list from the runtime and passes the arguments to the outlined parallel region (in the expected order).

Mapping OpenMP to GPUs

IBM

allocated in the
MASTER thread's
local memory by default,
BUT
must now be
"shareable"
with the WORKERS!

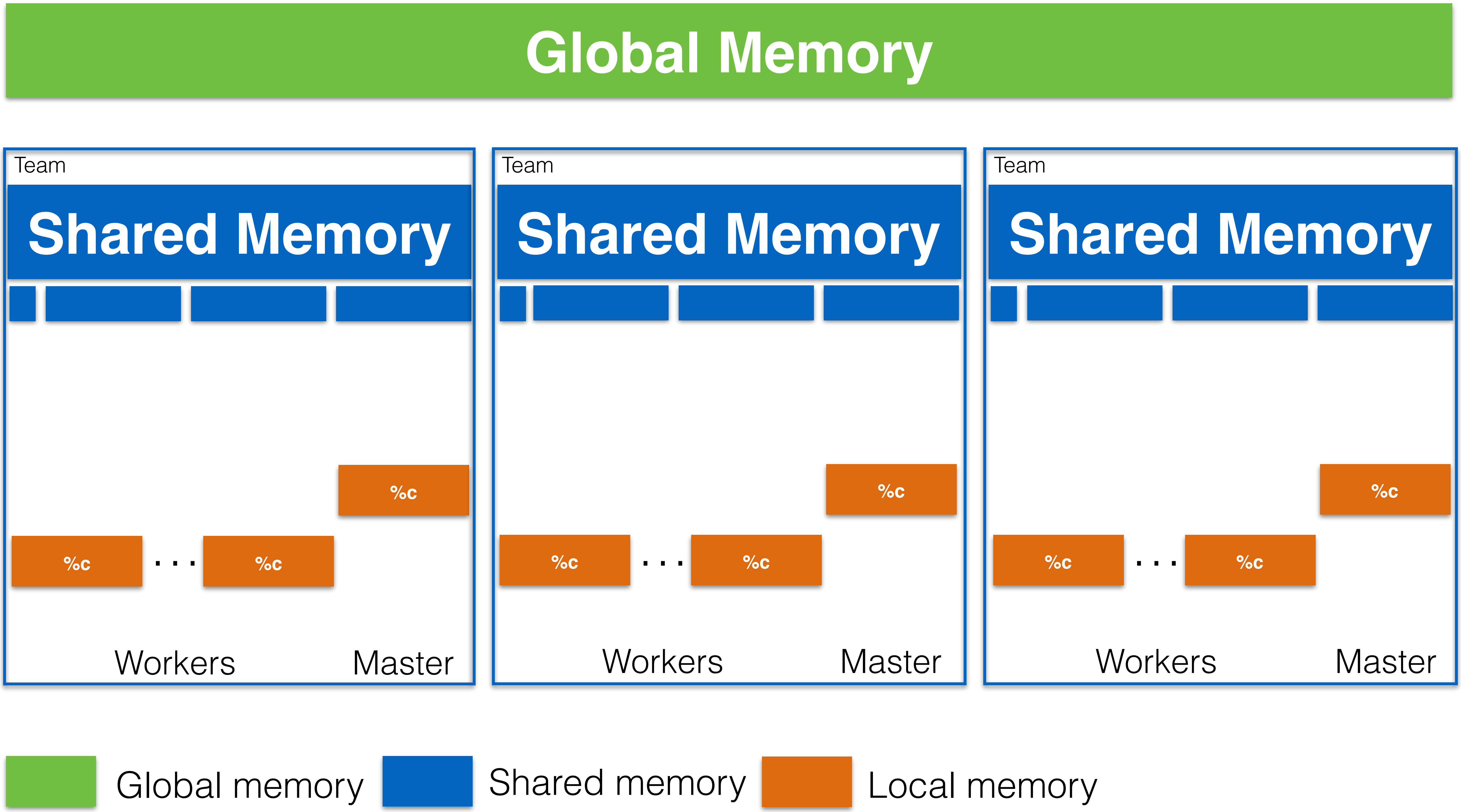
```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1; // LLVM-IR: %c = alloca i32
    #pragma omp parallel for
    for (i) {
      A[i] = c * i;
    }
    c += 2;
  }
}
```

1. In the CUDA model shared variables must be explicitly declared as __shared__.
2. On a GPU, variables allocated in local memory cannot be shared.

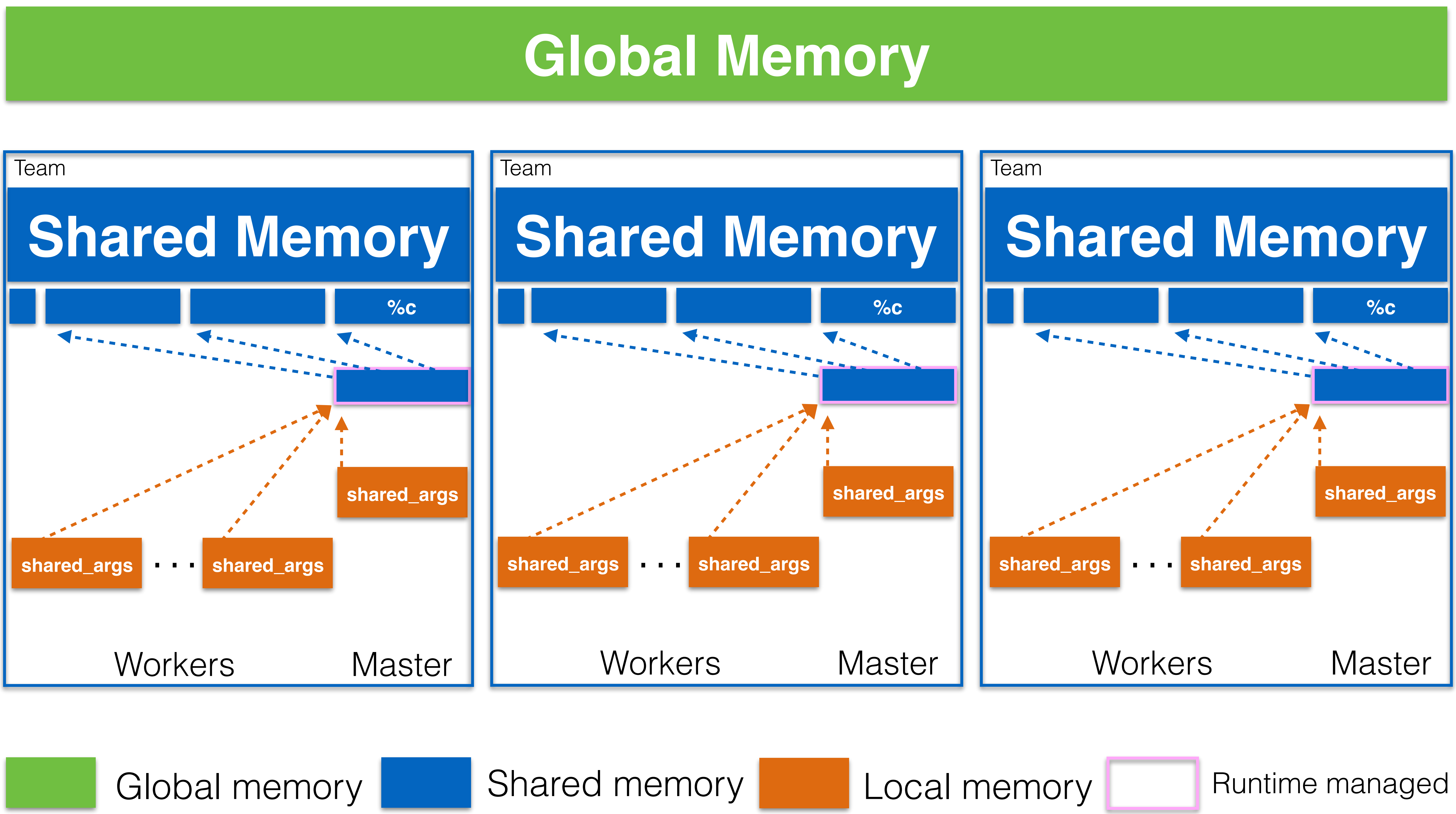
Changes to LLVM's NVPTX Backend

- There are 4 alternative ways for lowering a shared variable:
  - lower alloca to local memory - no sharing needed;
  - lower alloca to shared memory - one instance of the shared variable per team, store variable in shared memory stack, limited by shared memory size;
  - lower alloca to global memory - one instance per team but in global memory, no more team-level management of the variable, vulnerable to recursive functions;
  - lower alloca to runtime-managed memory - use a global memory stack managed by the runtime, supports all cases, interactions with runtime are expensive.

No Sharing


Global Memory
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Team
Shared Memory
%c
%c
. . .
%c
Workers
Master
Global memory
Shared memory
Local memory

Shared Memory Scheme



Change address space

IBM

- Introduce a new LLVM-IR pass which will recognize the cases where an alloca should use shared memory instead of local. Detection condition:
  - if an alloca has its address taken i.e. the alloca address is stored
- Insert two address space cast instructions from generic to shared and from shared to generic.

```
%a = alloca i32

…

store i32* %a, i32** %2
```

▶

```
%a = alloca i32
%1 = addrspacecast 0 to 3
%2 = addrspacecast 3 to 0

…

store i32* %a, i32** %2
```

Use a shared stack

- We need to change some of NVPTX's passes over the LLVM, Machine Instruction and PTX intermediate representations:
  - Introduce a new depot in the prologue of the kernel function for the allocation of shared variables.
  - Introduce a shared stack pointer which mimics the way the local stack pointer is set up in the entry block.

Use a shared stack

IBM

- Extend lowering of **alloca**'s to shared memory:
  - **SP** for **generic** address space operations.
  - **SPL** for **local** address space operations.
  - **SPSH** for **shared** address space operations.

```
kernel() {
 .local  .align 8 .b8 __local_depot[10]
 .shared .align 8 .b8 __shared_depot[10]

 mov.u64            %SPL, __local_depot
 mov.u64            %SPSH, __shared_depot
 cvta.local.u64     %SP, %SPL
 cvta.shared.u64    %SP, %SPSH

 add.u64            %rd1, %SPSH, 8
 ld.shared.u64      %rd2, [%rd1]
 …
}
```

PTX

Propagate use of shared stack pointer

- Add a new pass to the NVPTX that will lower the frame index of shared values to the shared stack pointer (SHSP).
- This pass operates on the internal representation of NVPTX (MI - Machine Instruction).

```
%vreg25<def> = LEA_ADDRi64 <fi#3>, 0;
%vreg6<def> = cvta_to_shared_yes_64 %vreg25<kill>;
```

MI - IR

```
%vreg25<def> = LEA_ADDRi64 %VRShared, 32;
```

MI - IR

Putting it all together

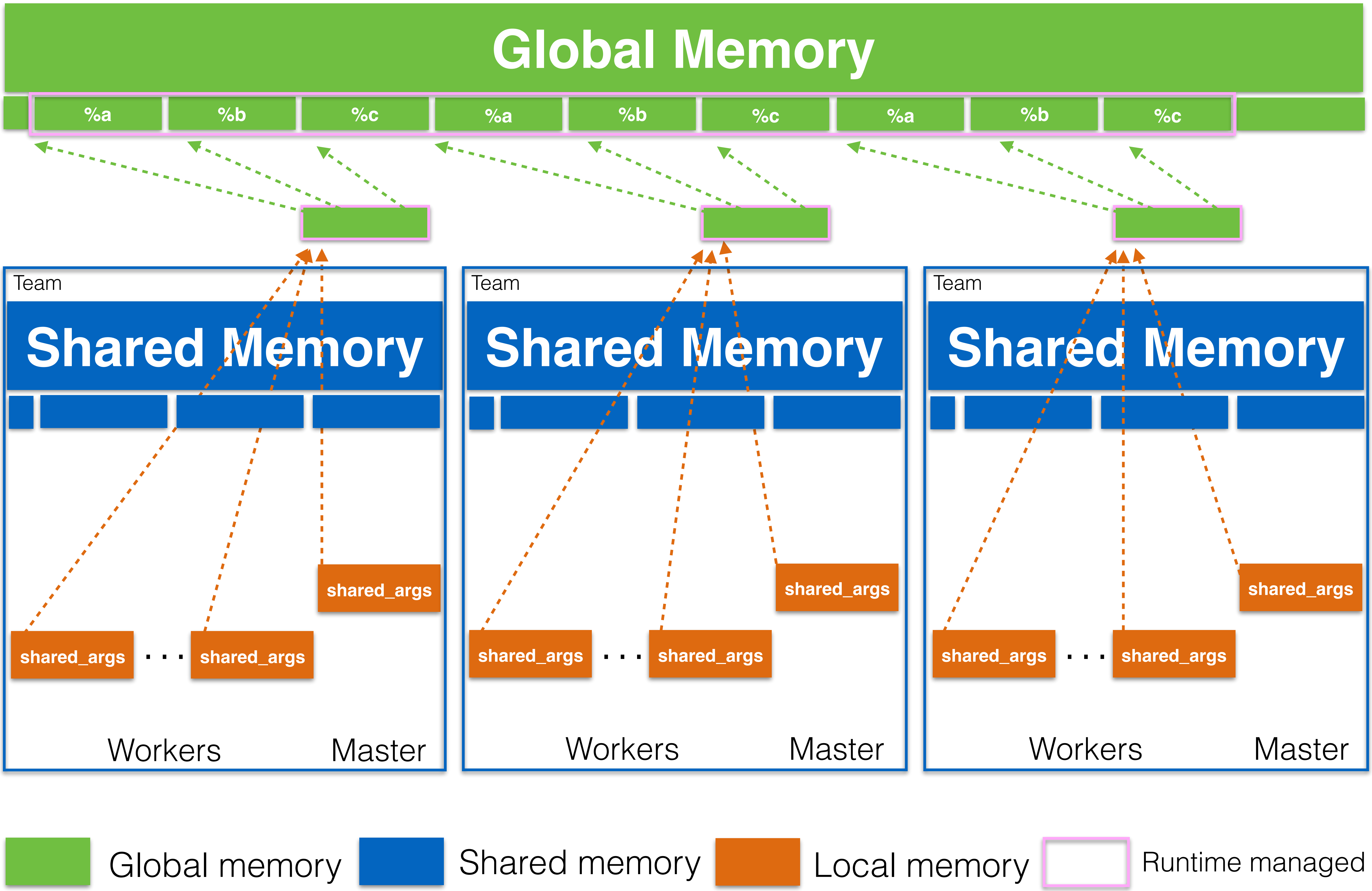
- Addition of a shared memory scheme compatible with the current code generation scheme:
  - we modified the runtime to share values from MASTER to WORKER threads.
  - we modified CLANG's code generation to support our data sharing convention.
- Sharing relies on variables being stored in a "shareable" memory address space on the device:
  - we modified LLVM's NVPTX Backend to support the lowering of shared variables to the GPU's shared memory.

Limitations & future work

- Limitations of the **new** data sharing scheme:
  - **No communication from CLANG to LLVM about OpenMP:** CUDA and OpenMP offloading share the same toolchain, distinguish between the two.
  - **Shared memory is limited:** adopt one of the more generic sharing alternatives for cases in which shared memory is insufficient or inefficient due to occupancy.
  - **Support for recursive functions**
  - **Support second level of sharing among WORKERS:** currently the new data sharing infrastructure only supports sharing from MASTER to WORKERS.
- These limitations do not apply to the current data sharing scheme.

Thank you for listening!
Questions?

Future Work: Global Memory Scheme



Future work: sharing among workers

```
void test(){
  int c = 5000;
  #pragma omp target
  {
    c += 1;
    #pragma omp parallel for
    for (i) {
      int d;
      d = c * i;
      #pragma omp simd
      for (j) {
         B[j] = d * j;
      }
    }
    c += 2;
  }
}
```